Introduction to Operating Systems

What's an OS?

The OS is a layer between applications and hardware to ease development.

- **Abstraction**. provides abstraction for applications:
  - manages + hides hardware details
  - uses low-level interfaces (not available to applications)
  - multiplexes hardware to multiple programs (*virtualization*)
  - makes hardware use efficient for applications
- **Protection**.
  - from processes using up all resources (*accounting*, *allocation*)
  - from processes writing into other processes memory
- **Resource Management**.
  - manages + multiplexes hardware resources
  - decides between conflicting requests for resource use
  - *goal*: efficient + fair resource use
- **Control**.
  - controls program execution
  - prevents errors and improper computer use

⇝ no universally accepted definition

Hardware Overview

- **Bus**: CPU(s)/devices/memory (conceptually) connected to common bus
  - CPU(s)/devices competing for memory cycles/bus
  - all entities run concurrently
  - *today*: multiple buses
- **Device controller**: has local buffer and is in charge of particular device
- **Interplay**:
  1. CPU issues commands, moves data to devices
  2. Device controller informs APIC that it has finished operation
  3. APIC signals CPU
  4. CPU receives device/interrupt number from APIC, executes handler

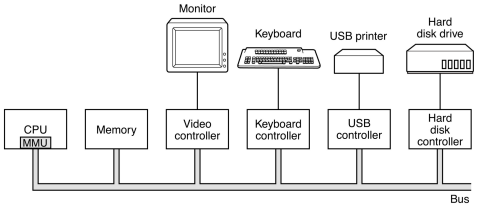


Figure 1: Traditional bus design.

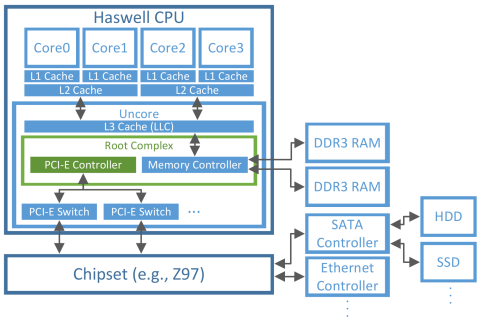


Figure 2: Modern bus design.

Central Processing Unit (CPU) — Operation

- **Principle**:
  1. *fetches* instructions from memory,
  2. *executes* them
- **During execution**: (meta-)data is stored in CPU-internal registers, i.e.
  - general purpose registers
  - floating point registers
  - instruction pointer (IP)
  - stack pointer (SP)
  - program status word (PSW)

CPU — Modes of Execution

- **User mode** (x86: *Ring 3*/*CPL 3*):
  - only non-privileged instructions may be executed
  - cannot manage hardware → *protection*
- **Kernel mode** (x86: *Ring 0*/*CPL 0*):
  - all instructions allowed
  - can manage hardware with *privileged instructions*

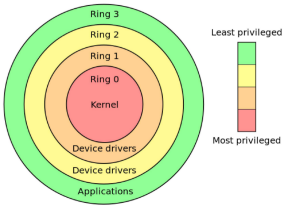


Figure 3: The different protection layers in the *ring model*.

Random Access Memory (RAM)

- **Principle**: keeps currently executed instructions + data
- **Connectivity**:
  - *today*: CPUs have built-in **memory controller**
  - *CPU caches*: "wired" to CPU
  - *RAM*: connected via pins
  - *PCI-E switches*: connected via pins

Caches

- **Problem**: RAM delivers instructions/data slower than CPU can execute
- **Locality principle**:
  - *spatial locality*: future refs often near previous accesses (e.g. next byte in array)
  - *temporal locality*: future refs often at previously accessed ref (e.g. loop counter)
- **Solution**: *caching* helps mitigating this memory wall
  1. *copy* used information temporarily from slower to faster storage
  2. *check* faster storage first before going down *memory hierarchy*
  3. if *not found*, data is copied to cache and used from there
- **Access latency**:
  - *register*: ~1 CPU cycle
  - *L1 cache* (per core): ~4 CPU cycles
  - *L2 cache* (per core pair): ~12 CPU cycles
  - *L3 cache/LLC* (per uncore): ~28 CPU cycles (~25 GiB/s)
  - *DDR3-12800U RAM*: ~28 CPU cycles + ~ 50ns (~12 GiB/s)

Device controlling

- **Device controller**: controls device, accepts commands from OS via *device driver*
- **Device registers/memory**:
  - *control* device by writing device registers
  - *read* status of device by reading device registers
  - *pass data* to device by reading/writing device memory
- **Device registers/memory access**:
  1. **port-mapped IO** (PMIO): use special CPU instructions to access port-mapped registers/memory
  2. **memory-mapped IO** (MMIO):
     - use same address space for RAM and device memory
     - some addresses map to RAM, others to different devices
     - access device's memory region to access device registers/memory
  3. **Hybrid**: some devices use hybrid approaches using both

Summary

- The OS is an **abstraction** layer between applications and hardware (multiplexes hardware, hides hardware details, provides protection between processes/users)
- The CPU provides a **separation** of User and Kernel mode (which are required for an OS to provide protection between applications)
- CPU can execute commands faster than memory can deliver instructions/data — memory **hierarchy** mitigates this memory wall, needs to be carefully managed by OS to minimize slowdowns
- device drivers **control** hardware devices through PMIO/MMIO
- Devices can **signal** the CPU (and through the CPU notify the OS) through interrupts

OS Concepts

OS Invocation

- OS Kernel does **not** always run in background!
- Occasions invoking kernel, switching to kernel mode:
  1. **System calls**: User-Mode processes require higher privileges
  2. **Interrupts**: CPU-external device sends signal
  3. **Exceptions**: CPU signals unexpected condition

System Calls — Motivation

- **Problem**: protect processes from one another
- **Idea**: Restrict processes by running them in user-mode
- ⇝ **Problem**: now processes cannot manage hardware,...
  - who can switch between processes?
  - who decides if process may open certain file?
- ⇝ **Idea**: OS provides **services** to apps
  1. app calls system if service is needed (**syscall**)
  2. OS checks if app is allowed to perform action
  3. if app may perform action and hasn't exceeded quota, OS performs action in behalf of app in kernel mode

System Calls — Examples

- `fd = open(file, how,...)` – open file for read/write/both
- documented e.g. in `man 2 write`
- overview in `man 2 syscalls`

System Calls vs. APIs

- **Syscalls**: interface between apps and OS services, limited number of well-defined entry points to kernel
- **APIs**: often used by programmers to make syscalls (e.g. `printf` library call uses `write` syscall)
- common APIs: Win32, POSIX, C API

System Calls — Implementation

- **Trap Instruction**: single syscall interface (entry point) to kernel
  - switches CPU to kernel mode, enters kernel in same way for all syscalls
  - *system call dispatcher* in kernel then acts as syscall multiplexer
- **Syscall Identification**: number passed to trap instruction
  - *Syscall Table* maps syscall numbers to kernel functions
  - *Dispatcher* decides where to jump based on number and table
  - programs (e.g. `stdlib`) have syscall number compiled in!

  ⇝ never reuse old syscall numbers in future kernel versions

Interrupts

- **Devices**: use interrupts to signal predefined conditions to OS
  - *reminder*: device has "interrupt line" to CPU (e.g. device controller informs CPU that operation is finished)
- **Programmable Interrupt Controller**: manages interrupts
  - interrupts can be *masked* (queued, delivered when interrupt unmasked)
  - queue has finite length ⇝ interrupts can get lost
- **Examples**:
  1. *timer-interrupt*: periodically interrupts processes, switches to kernel ⇝ can then switch to different processes for fairness
  2. *network interface card* interrupts CPU when packet was received ⇝ can deliver packet to process and free NIC buffer
- **Interrupt process**:
  1. CPU looks up *interrupt vector* (= table pinned in memory, contains addresses of all service routines)
  2. CPU transfers control to respective *interrupt service routine* in OS that handles interrupt

  ⇝ interrupt service routine must first save interrupted process's state (instruction pointer, stack pointer, status word)

Exceptions

- **Motivation**: unusual condition → impossible for CPU to continue processing
- ⇝ **Exception** generated within CPU:
  1. CPU interrupts program, gives kernel control
  2. kernel determines reason for exception
  3. if kernel can resolve problem ⇝ does so, continues *faulting instruction*
  4. kills process if not
- **Difference to Interrupts**: interrupts can happen in any context, exceptions always occur asynchronous and in process context

OS Concepts — Physical Memory

- up to early 60s:
  - programs loaded and run directly in *physical memory*
  - program too large → partitioned manually into *overlays*
  - OS: swaps overlays between disk and memory
  - different jobs could observe/modify each other

OS Concepts — Address Spaces

- **Motivation**: bad programs/people need to be isolated
- **Idea**: give every job the illusion of having all memory to itself
  - every job has own *address space*, can't name addresses of others
  - jobs always and only use virtual addresses

Virtual Memory — Indirect Addressing

- **MMU**: every CPU has built-in *memory management unit* (MMU)
- **Principle**: translates virtual addresses to physical addresses at every load/store

  ⇝ address translation protects one program from another
- **Definitions**:
  - *Virtual address*: address in process' address space
  - *Physical address*: address of real memory

Virtual Memory — Memory Protection

- **Kernel-only Virtual Addresses**
  - kernel typically part of all address spaces
  - ensures that apps can't touch kernel memory
- **Read-only virtual addresses**: can be enforced by MMU
  - allows safe sharing of memory between apps
- **Execute Disable**: can be enforced by MMU
  - makes code injection attacks harder

Virtual Memory — Page Faults

- **Motivation**: not all addresses need to be mapped at all times
  - MMU issues *page fault* exception when accessed virtual address isn't mapped
  - OS handles page faults by loading faulting addresses and then continuing the program

  ⇝ memory can be *over-committed*: more memory than physically available can be allocated to application
- **Illegal addresses**: page faults also issued by MMU on illegal memory accesses

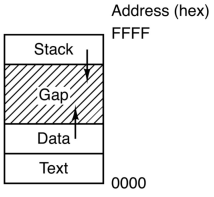
OS Concepts — Processes

- **Process**: program in execution ("instance" of program)
- each process is associated with
  - **Process Control Block** (PCB): contains information about allocated resources
  - virtual **Address Space** (AS):
    - all (virtual) memory locations a program can name
    - starts at 0 and runs up to a maximum
    - address 123 in AS1 generally ≠ address 123 in AS2
    - indirect addressing ⇝ different ASes to different programs

  ⇝ *protection between processes*

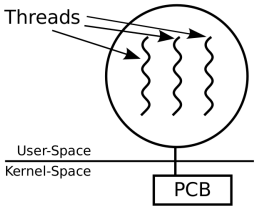
OS Concepts — Address Space Layout

- **Sections**: address spaces typically laid-out in different sections
  - memory addresses between sections *illegal*
  - illegal addresses ⇝ page fault (*segmentation fault*)
  - OS usually kills process causing segmentation fault
- **Important sections**:
  - *Stack*: function history, local variables
  - *Data*: Constants, static/global variables, strings
  - *Text*: Program code



OS Concepts — Threads

- **Thread**: represents execution state of process (≥ 1 thread per process)
  - *IP*: stores currently executed instruction (address in `text` section)
  - *SP*: stores address of stack top (> 1 threads → multiple stacks!)
  - *PSW*: contains flags about execution history (e.g. last calculation was 0 → used in following jump instruction)
  - more general purpose registers, floating point registers,...



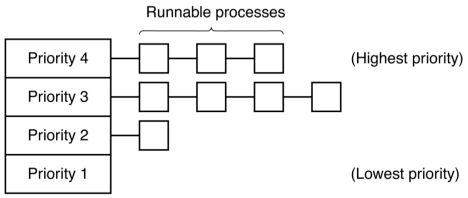
OS Concepts — Policies vs. Mechanisms

- **Mechanism**: implementation of what is done (e.g. commands to write to HDD)
- **Policy**: rules which decide when what is done and how much (e.g. how often, how many resources are used,...)

→ *mechanisms can be reused even when policy changes*

OS Concepts — Scheduling

- **Motivation**: multiple processes/threads available ⇝ OS needs to switch between them (for multitasking)
- **Scheduler**: decides which job to run next (*policy*) — tries to
  - provide fairness
  - meet performance goals
  - adhere to priorities
- **Dispatcher**: performs task-switching (*mechanism*)



OS Concepts — Files

- **Motivation**: OS hides peculiarities of file storage, programmer uses device-independent *files/directories*
- **Files**: associate *file name* and *offset* with bytes
- **Directories**: associate *directory names* with directory names or file names
- **File System**: ordered block collection
  - main task: translate (dir name + file name + offset) to block
  - programmer uses file system operations to operate on files (`open`, `read`, `seek`)
  - processes can communicate directly through special *named pipe* file (used with same operations as any other file)

OS Concepts — Directory Tree

- **Directories**: form *directory tree/file hierarchy* → structure data
- **Root Directory**: topmost directory in tree
- **Path Name**: used to specify file

OS Concepts — Mounting

- **Unix**: common to orchestrate multiple file systems in single file hierarchy
- file systems can be *mounted* on directory
- **Win**: manage multiple directory hierarchies with drive letters (e.g. `C:\Users`)

OS Concepts — Storage Management

- **OS**: provides uniform view of information storage to file systems
  - *Drivers*: hide specific hardware devices → hides device peculiarities
  - general interface abstracts physical properties to logical units → block
- **Performance**: OS increases I/O performance:
  - *Buffering*: Store data temporarily while transferred
  - *Caching*: Store data parts in faster storage
  - *Spooling*: Overlap one job's output with other job's input

Summary

- **OS**: provides abstractions for and protection between applications
- **Kernel**: does not always run — certain events invoke kernel
  - *syscall*: process asks kernel for service
  - *interrupt*: device sends signal that OS has to handle
  - *exception*: CPU encounters unusual situation
- **Processes**: encapsulate resources needed to run program in OS
  - *threads*: represent different execution states of process
  - *address space*: all memory process can name
  - *resources*: allocated resources, e.g., open files
- **Scheduler** decides which process to run next when multi-tasking
- **Virtual Memory** implements address spaces, provides protection between processes
- **File system** abstracts background store using I/O drivers, provides simple interface (files + directories)

Processes

The Process Abstraction

- **Motivation**: computers (seem to) do "several things at the same time" (quick process switching → *multiprogramming*)
- **Model**: *process abstraction* models this concurrency:
  - container contains information about program execution
  - conceptually, every progress has own "'virtual CPU"'
  - execution context is changed on process switch
  - dispatcher switches context when switching processes
  - **context switch**: dispatcher saves current registers/memory mappings, restores those of next process

Process-Cooking Analogy

- Program/Process like Recipe/Cooking
- **Recipe**: lists ingredients, gives algorithm what to do when
  ⇝ program describes memory layout/CPU instructions
- **Cooking**: activity of using the recipe
  ⇝ process is activity of executing a program
- multiple similar recipes for same dish
  ⇝ multiple programs may solve same problem
- recipe can be cooked in different kitchens at the same time
  ⇝ program can be run on different CPUs at the same time (as different processes)
- multiple people can cook one recipe
  ⇝ one process can have several worker threads

Concurrency vs. Parallelism

- OS uses currency + parallelism to implement multiprogramming
  1. **Concurrency**: multiple processes, one CPU
     ⇝ not at the same time
  2. **Parallelism**: multiple processes, multiple CPU
     ⇝ at the same time

Virtual Memory Abstraction — Address Spaces

- every process has own *virtual addresses* (`vaddr`)
- MMU relocates each load/store to *physical memory* (`pmem`)
- processes never see physical memory, can't access it directly
+ MMU can enforce protection (mappings in kernel mode)
+ programs can see more memory than available
  - 80:20 rule: 80% of process memory idle, 20% active
  - can keep working set in RAM, rest on disk
- need special MMU hardware

Address Space (Process View)

- **Motivation**: code/data/state need to be organized within process
  ⇝ *address space layout*
- **Data types**:
  1. *fixed size* data items
  2. data naturally *freed in reverse allocation order*
  3. data *allocated/freed "'randomly"'*
- compiler/architecture determine how large int is and what instructions are used in text section (`code`)

- **Loader** determines based on exe file how executed program is placed in memory

Segments — Fixed-Size Data + Code

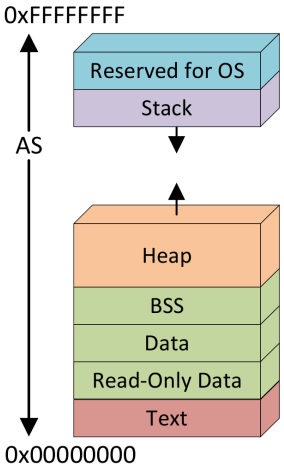
- some data in programs never changes or will be written but never grows/shrinks
  ⇝ memory can be statically allocated on process creation
- **BSS segment** (*block started by symbol*):
  - statically allocated variables/non-initialized variables
  - executable file typically contains starting address + size of BSS
  - entire segment initially 0
- **Data segment**: fixed-size, initialized data elements (e.g. global variables)
- **Read-only data segment**: constant numbers, strings
- All three sometimes summarized as one segment
- compiler and OS decide ultimately where to place which data/how many segments exist

Segments — Stack

- some data naturally freed in reverse allocation order
  - very easy memory management (stack grows upwards)
- fixed segment starting point
- store top of latest allocation in **stack pointer** (SP) (initialized to starting point)
- *allocate* `a` byte data structure: `SP += a; return(SP - a)`
- *free* `a` byte data structure: `SP -= a`

Segments — Heap (Dynamic Memory Allocation)

- some data "randomly" allocated/freed
- two-tier memory allocation:
  1. allocate large memory chunk (**heap segment**) from OS
     - base address + **break pointer** (BRK)
     - process can get more/give back memory from/to OS
  2. dynamically partition chunk into smaller allocations
     - `malloc`/`free` can be used in random oder
     - purely user-space, no need to contact kernel



Summary

Processes: recipe vs. cooking = program vs. process
- processes = resource container for OS
- process feels alone (has own CPU and memory)
- OS implements multiprogramming through rapid process switching

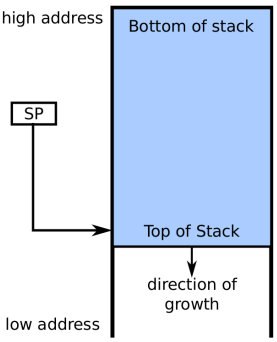
Process API

Execution Model — Assembler (simplified)

- **Principle**: OS interacts directly with compiled programs
  - switch between processes/threads ⇝ *save/restore* state
  - deal with/pass on *signals/exceptions*
  - receive *requests* from applications
- **Instructions**:
  - `mov`: Copy referenced data from second operand to first operand
  - `add`/`sub`/`mul`/`div`: Add,…from second operand to first operand
  - `inc`/`dec`: increment/decrement register/memory location
  - `shl`/`shr`: shift first operand left/right by amount given by second operand
  - `and`/`or`/`xor`: calculate bitwise and,…of two operands storing result in first
  - `not`: bitwise negate operand

Execution Model — Stack (x86)

- **stack pointer** (SP): holds address of stack top (growing downwards)
- **stack frames**: larger stack chunks
- **base pointer** (BP): used to organize stack frames



Execution Model — jump/branch/call commands (x86)

- `jmp`: continue execution at operand address
- `j$condition`: jump depending on PSW content
  - *true* ⇝ jump
  - *false* ⇝ continue
  - examples: `je` (jump equal), `jz` (jump zero)
- `call`: push function to stack and jump to it
- `return`: return from function (jump to return address)

Execution Model — Application Binary Interface (ABI)

- **Idea**: standardizes binary interface between programs, modules, OS:
  - executable/object file layout
  - calling conventions
  - alignment rules
- **calling conventions**: standardize exact way function calls are implemented
  ⇝ interoperability between compilers

Execution Model — calling conventions (x86)

- function call — **caller**:
  1. save local scope state
  2. set up parameters where function can find them
  3. transfer control flow
- function call — **called function**:
  1. set up new local scope (local variables)
  2. perform duty
  3. put return value where caller can find it
  4. jump back to caller (IP)

Passing parameters to the system

- parameters are passed through **system calls**
- call number + specific parameters must be passed
- parameters can be transferred through
  - **CPU registers** (~6)
  - **Main Memory** (heap/stack – more parameters, data types)
- ABI specifies how to pass parameters
- **return code** needs to be returned to application
  - *negative*: error code
  - *positive + 0*: success
  - usually returned via A+D registers

System call handler

- implements the actual service called through a syscall:
  1. saves tainted registers
  2. reads passed parameters
  3. sanitizes/checks parameters
  4. checks if caller has enough permissions to perform the requested action
  5. performs requested action in behalf of the caller
  6. returns to caller with success/error code

Process API — creation

- process creation events:
  1. system initialization
  2. process creation syscall
  3. user requests process creation
  4. batch job-initiation
- events map to two mechanisms:
  1. Kernel spawns initial user space process on boot (Linux: `init`)
  2. User space processes can spawn other processes (within their quota)

Process API — creation (POSIX)

- `PID`: identifies process
- `pid = fork()`: duplicates current process:
  - returns 0 to new child
  - returns new `PID` to parent

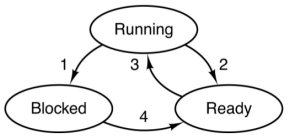
  ⇝ child and parent independent after `fork`
- `exec(name)`: replaces own memory based on executable file
  - `name` specifies binary executable file
- `exit(status)`: terminates process, returns `status`
- `pid = waitpid(pid, &status)`: wait for child termination
  - `pid`: process to wait for
  - `status`: points to data structure that returns information about the process (e.g., exit status)
  - passed `pid` is returned on success, `-1` otherwise
- **process tree**: processes create child processes, which create child processes, …
  - parent and child execute concurrently
  - parent waits for child to terminate (collecting the exit state)

Daemons

= program designed to run in the background
- detached from parent process after creation, reattached to process tree root (`init`)

Process States

- **blocking**: process does nothing but wait
  - usually happens on syscalls (OS doesn't run process until event happens)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process Termination

- different termination events:
  1. normal exit (voluntary)
     - `return 0` at end of `main`
     - `exit(0)`
  2. error exit (voluntary)
     - `return x` (`x` ≠ 0) at end of `main`
     - `exit(x)` (`x` ≠ 0)
     - `abort()`
  3. fatal error (involuntary)
     - OS kills process after exception
     - process exceeds allowed resources
  4. killed by another process (involuntary)
     - another process sends kill signal (only as parent process or administrator)

Exit Status

- voluntary exit: process returns exit status (integer)
- resources not completely freed after process terminates ⇝ **Zombie** or **process stub** (contains exit status until collected via `waitpid`)
- **Orphans**: Processes without parents
  - usually adopted by `init`
  - some systems kill all children when parent is killed
- exit status on involuntary exit:
  - Bits `0-6`: signal number that killed process (0 on normal exit)
  - Bit 7: set if process was killed by signal
  - Bits `8-15`: 0 if killed by signal (exit status on normal exit)

Threads

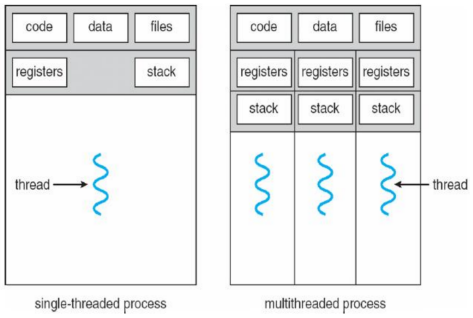
Processes vs. Threads

- **Traditional OS**: each process has
  - own address space
  - own set of allocated resources
  - *one* thread of execution (= one execution state)
- **Modern OS**: processes + threads of execution handled more flexibly
  - *processes* provide abstraction of address space and resources
  - *threads* provide abstraction of execution states of that address space
- **Exceptions**:
  - sometimes different threads have different address spaces
  - *Linux*: threads = regular processes with shared resources and AS regions

Threads — why?

- many programs do multiple things at once (e.g. web server)

  ⇝ writing program as many sequential threads may be easier than with blocking operations
- **Processes**: rarely share data (if, then explicitly)
- **Threads**: closely related, share data



Threads — POSIX

- `PThread`: base object with
  - *identifier* (thread ID, TID)
  - *register set* (including IP and SP)
  - *stack area* to hold execution state
- `Pthread_create`: create new thread
  - Pass: *pointer* to `pthread_t` (will hold TID after successful call)
  - Pass: *attributes*, *start function*, *arguments*
  - Returns: 0 on success, error value else
- `Pthread_exit`: terminate calling thread
  - Pass: exit code (casted to void pointer)
  - Free's resources (e.g. stack)
- `Pthread_join`: wait for specified thread to exit
  - Pass: `ptread_t` to wait for (or `-1` for any thread)
  - Pass: pointer to pointer for exit code
  - Returns: 0 on success, error value else
- `Pthread_yield`: release CPU to let another thread run

Threads — Problems

- **Processes vs. Threads**:
  - *Processes*: only share resources explicitly
  - *Threads*: more shared state → more can go wrong
- **Challenges**: programmer needs to take care of
  - *activities*: dividing, ordering, balancing
  - *data*: dividing
  - *shared data*: access synchronizing

PCB vs. TCP

- **PCB** (*process control block*): information needed to implement processes
  - always known to OS
- **TCB** (*thread control block*): per thread data
  - OS knowledge depends on *thread model*

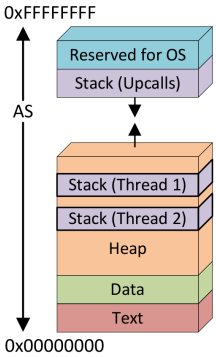
PCB	TCB
Address space	Instruction pointer
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	

Thread models

- **Kernel Thread**: known to OS kernel
- **User Thread**: known to process
- **N:1-Model**: kernel only knows one of possibly multiple threads
  - N:1 user threads = *user level threads* (ULT)
- **1:1-Model**: each user thread maps to one kernel thread
  - 1:1 user threads = *kernel level threads* (KLT)
- **M:N-Model** (hybrid model): flexible mapping of user threads to less kernel threads

Thread models — N:1

- Kernel only manages process → multiple threads unknown to kernel
- Threads managed in user-space library (e.g. GNU Portable Threads)
- **Pro**:
  + faster thread management operations (up to 100 times)
  + flexible scheduling policy
  + few system resources
  + usable even if OS doesn't support threads
- **Con**:
  - no parallel execution
  - whole process blocks if one user thread blocks
  - reimplementing OS parts (e.g. scheduler)
- **Stack**:
  - main stack known to OS used by thread library
  - own execution state (= stack) dynamically allocated by user thread library for each thread
  - possibly own stack for each exception handler
- **Heap**:
  - concurrent heap use possible
  - *Attention*: not all heaps are reentrant
- **Data**: divided into BSS, data and read-only data here as well

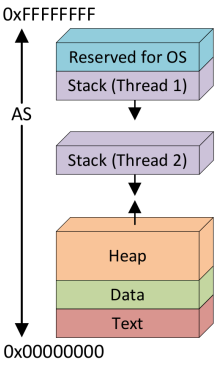


Thread models — 1:1

- kernel knows + manages every thread
- **Pros**:
  + real parallelism possible
  + threads block individually
- **Cons**:
  - OS manages every thread in system (TCB, stacks,…)
  - Syscalls needed for thread management
  - scheduling fixed in OS
- **Stack**:
  - own execution state (= stack) for every thread
  - possibly own stack for (each) exception handler
- **Heap**:
  - parallel heap use possible
  - *Attention*: not all heaps are thread-safe
  - if thread-safe: not all heap implementations perform well with many threads
- **Data**: divided into BSS, data and read-only data here as well

Thread models — M:N

- **Principle**: M ULTs are maps to (at most) N KLT
  - *Goal*: pros of ULT and KLT — non-blocking with quick management
  - create sufficient number of KLTs and flexibly allocate ULTs to them



  - *Idea*: if ULT blocks ULTs can be switched in userspace
- **Pros**:
  + flexible scheduling policy
  + efficient execution
- **Cons**:
  - hard to debug
  - hard to implement (e.g. blocking, number of KLTs,…)
- **Implementation — Up-calls**:
  - kernel notices that thread will block → sends signal to process
  - up-call notifies process of thread id and event that happened
  - exception handler of process schedules a different process thread
  - kernel later informs process that blocking event finished via other up-call

Summary

- programs often do closely related things at once
  - mapped to thread abstraction: multiple threads of execution operate in same process
- differentiation between process information (PCB) and thread information (TCB)
- **thread models**:
  - $N:1$: threads fully managed in user-space
  - $1:1$: threads fully managed by kernel
  - $M:N$: threads are flexibly managed either in user-space or kernel
- multi-threaded programs operate on same data concurrently or even parallel:
  - *synchronization*: accessing such data must be synchronized

  → makes writing such programs challenging